Parallelization of File Descriptor Management

Tera Computer Company
400 N. 34th St.
Seattle, WA 98103

June 4, 1993

1 Introduction

When a task opens a file, the operating system returns a file descriptor and provides an index into a pool of file structure pointers referenced from the task's file descriptor table. Each task has its own file descriptor table.

Under *BSD4.4* the size of a pool of file structure pointers may grow during runtime bounded by an upper limit. During the time when the pointer list is expanded, all activities must be suspended till the old list has been copied into the new list and all future references are directed to the new list.

A file structure may be pointed to from one or more file descriptor tables as a result of a fork, dup, or dup2 system call.

There is a chain list of active files maintained within the entire system. This list is accessed by a variable named *filehead* which points to file structures that are linked via *f_filef* and *f_fileb* pointers. This list is used in particular by the *unp_gc()* routine for garbage collection of lost references to files that are present in the Unix domain socket.

2 File Descriptor Table

In parallelizing accesses to a task's file descriptor table, a simple spin lock named *fd_spLock* is used to protect all fields in the *filedesc* structure. In particular, *fd_spLock* is used to protect the allocation and deallocation of a file structure pointer.

A pointer is added into the *filedesc* structure that addresses a list of counters for all file descriptors. This pointer named *fd_othreadCount* addresses pointers each of which is used to keep count of the number of threads that are actively doing io on a specific file descriptor. A thread that calls *close()* will wait for all threads that are currently doing io to complete by sleeping on a global sleep channel named *closeSleepChan*. The last thread that accesses this file descriptor will wake up the thread that is sleeping in the *closeSleepChan* sleep channel.

A global sleep channel named *openSleepChan* is used by all system calls that reference a specified file descriptor to wait until a file has completely opened. Basically, flags in *fd_ofileflags* are set to *UF_OPENING* or *UF_WAITOPEN* to indicate that a file descriptor is in the process of an *open* or that a thread is waiting for *open* to complete, respectively.

A global sleep channel named *fd_allocatingSleepChan* is used to synchronize the expansion of a file descriptor table and will be further described in the section on allocation of a file structure.

3 File Structure

First, a simple spin lock named *f_spLock* is used to protect all accesses for the following members in a *file* structure. These members include *f_count, f_flag, f_msgcount, f_cred, f_offset*, and *f_data.* The rest of these members are set at allocation, but are never altered until deallocation. These members include: *f_type*, and *f_ops.*

The system's active file linked list fields *f_filef* and *f_fileb* are manipulated via a global spin lock named *activeFile_spLock.* This is discussed further in a later section on active file list.

A file's offset *f_offset* is accessed via a spin suspend lock named *f_spinSuspendLock.* This essentially serializes *read, write*, and *lseek* accesses to the same file descriptor and provides consistency to a file's offset for Unix *read, write*, and *lseek* system calls.

A global sleep channel named *uf_allocatingSleepChan* is used to synchronize the allocation of a new file descriptor *file* structure. Further description can be found in the next section.

4 Allocation of a File Descriptor

To allocate a new file descriptor requires first reserving a free file structure pointer in the file descriptor table by calling *fdalloc().* If a file descriptor table is full and is still below the current allowable limit the list of file structure pointers will be expanded. To avoid other threads from waiting during the expansion of the table's pointer list, a new flag named *fd_flag* is set to *FD_ALLOCATING* to let other *fdalloc()* calls know that a current expansion is ongoing. The caller then gives up the *fd_spLock* and calls the potentially blocking *MALLOC* function. On return, it wakes up all threads that are waiting on the *fd_allocatingSleepChan* such as a caller of *dup2().*

After an *index* is reserved in the file descriptor table, a file structure is allocated via *MALLOC* and inserted into the reserved *fd_ofiles[index]* location. In order to avoid races with the *dup2* system call, the specific *fd_ofileflags* is set to *UF_ALLOCATING.* On return from *MALLOC,* it wakes up all threads that are waiting on *uf_allocatingSleepChan.*

5 Deallocation a File Descriptor

When a task closes a file, a file descriptor pointer would be marked NULL in the file descriptor table and the file structure which it points to may become deallocated. When *closef()* is called, it checks whether the *fd_othreadCount* for the file descriptor is zero and if so proceed with the actual close. Otherwise, it sleeps on the global *closeSleepChan* until the thread count for the descriptor goes to zero. This ensures all threads that are currently using the file descriptor to complete before *close* can proceed. At the point when all threads within a task have completed their io on the particular file descriptor *close* then checks the task count *fd_fcount.* If it is zero this indicates all tasks accessing this file descriptor have completed and that it is safe to free the memory for the file descriptor.

In order to guarantee that no other threads are going to do further io on the same file descriptor, *close* must first set *fd_ofiles* for the file descriptor to *NULL*. This will ensure all future system calls that access the file descriptor to return an error at examining its *fd_ofiles* field. In addition, we must prevent a file descriptor from being freed when io is ongoing within the same file descriptor. This is ensured by requiring every system call that accesses its file descriptor fields to increment its thread count in *fd_othreadCount*. In fact, every system call that accesses a file structure must increment *fd_othreadCount* at the beginning of the call and decrement it before returning to caller. At the time of decrementing *fd_othreadCount*, it is the responsibilty of each caller to check and see if *fd_othreadCount* has gone to zero, and if so the caller must wakeup those sleeping on the *closeSleepChan* sleep channel.

The following listing consists of code for incrementing and decrementing *fd_othreadCount* on entry and exit to system entry routines that access its file descriptor table.

```
/* Parallel: lock file descriptor table, then increment
 *              thread count for the file descriptor.
 *          If successful returns 0 and file structure
 *              pointer.   Note: if returned successfully
 *              fd_spLock will be acquired after the call.
 *          Otherwise, returns EBADF, file structure ptr
 *              is NULL and fd_spLock is released.
 */
fd_checkCountLock(struct filedesc * fdp, int i, vfileptr *fp)
{
        Spin_lock((SpinLock_k *)&fdp->fd_spLock);
        if ((unsigned) i >= fdp->fd_nfiles ||
            (*fp = fdp->fd_ofiles[i]) == NULL) {
                Spin_unlock((SpinLock_k *)&fdp->fd_spLock);
                return (EBADF);
        }
        assert_k(fdp->fd_othreadCount[i] >= 0);
        fdp->fd_othreadCount[i]++;
        return (0);
}

/* Parallel: lock file descriptor table, then decrement
 *              thread count for the file descriptor.     If
 *              thread count goes to 0 wakeup who may be in closef
 */
int fd_countUnlock(struct filedesc *fdp, int i)
{

        if (!Spin_holdingLock((SpinLock_k *)&fdp->fd_spLock))
                Spin_lock((SpinLock_k *)&fdp->fd_spLock);
        assert_k(fdp->fd_othreadCount[i] > 0);
        fdp->fd_othreadCount[i]--;
        if (fdp->fd_othreadCount[i] <= 0)
                wakeup(&closeSleepChan);
        Spin_unlock((SpinLock_k *)&fdp->fd_spLock);
        return (0);
}
```

6 Active File List

A global spin lock named *activeFile_spLock* is used to protect the entire system's active file list. The primary users of this spin lock is the open and close system call, and the uipc garbage collector.

Since this global spin lock could become a source of contention for *open* and *close* system calls, the *mark-and-sweep* algorithm in *unp_gc()* must be revisited in the future.

In the *unp_gc()* function *activeFile_spinLock* must be taken since *filehead, f_filef*, and *f_fileb* are examined. However, *unp_gc* assumes that while examining each file structure in the list that each entry will not be deallocated by *ffree()* since this is guaranteed by a nonpreemptive kernel. The *f_spLock* must be taken prior to manipulatin *f_flag*. Note that *unp_gc* only calls *closef()* if and only if *f_count* is not zero.

To prevent a file descriptor from being freed when we are in *unp_gc* we first take the *activeFile_spLock*, and then we lock the file structure's *f_spLock* guaranteeing *f_count* to remain unchanged. If and when *f_count* is zero then *unp_gc* do not take any action with the file structure examined. In fact, *unp_gc* calls *closef()* if and only if *f_count* is not zero. We need to revisit *unp_gc()* in the future!

7 Hierarchy of Locks

The following lock hierarchy must be observed in order to avoid deadlock.

1. *fd_spLock* has precedence over *activeFile_spLock*
2. *activeFile_SpLock* has precedence over *f_spLock*
3. *fd_spLock* has precedence over *f_spLock*
4. *fd_spLock* has precedence over *f_spSuspendLock*

8 Future Parallelization Improvements

The system's active file list used specifically for uipc garbage collection must be redesigned to prevent a single source of contention. A possible solution is to keep a separate active file list for all files that have *f_type* equal to *DTYPE_SOCKET*.

The expansion of the file structure pointers list requires serializing accesses to all file descriptors. To eliminate this bottleneck, we may want to eliminate expansion all togther or design a different scheme for list expansion, such as use a forwarding bit for the last entry in the first list and so on.